Problem Set #1 Sample Solutions

Greg Plaxton

February 11, 2019

1(a). Write ℓ as $2^k + r$, where $0 \leq r < 2^k$. We claim that an ℓ-leaf binary tree with minimum external path length has $2r$ leaves at depth $k+1$ and $\ell - 2r$ leaves at depth k (the root is at depth 0). We prove this claim in two stages. First, we argue that any ℓ-leaf binary tree with minimum external path length is *full*, that is, every internal node has degree two. This is easy to see since the external path length of a non-full ℓ-leaf binary tree can be strictly reduced by removing a degree-one internal node, replacing it with its child. We now complete the proof of the claim by arguing that any full ℓ-leaf binary tree with minimum external path length has all of its leaves on at most two adjacent levels. To see this, consider a full ℓ-leaf binary tree with deepest leaf x at depth d and shallowest leaf y at level d' where $d' < d - 1$. Because the tree is full and x is a deepest leaf, x has a sibling leaf x', also at depth d. Let p denote the parent of leaves x and x'. Consider modifying this tree by making x and x' children of y instead of p. The resulting tree still has ℓ leaves: The only changes to the set of leaves are that p has been added and y has been removed. The change in the external path length is easy to calculate: There is a decrease of $2(d - d' - 1)$ due to the change in the depths of x and x', and there is an increase of $d - d' - 1$ due to the replacement of y by p in the set of leaves, for a net decrease of $d - d' - 1$, which is a positive quantity. This completes the proof of the claim. The desired $\Omega(\ell \lg \ell)$ lower bound now follows easily.

An alternative proof uses (strong, i.e., course of values) induction on ℓ, where the key technical observation needed to carry out the induction step is that the function $f(x) = x \lg x$ is convex (positive second derivative), so that an expression such as $f(a) + f(\ell - a)$ is minimized when $a = \frac{\ell}{2}$.

1(b). Fix a comparison-based sorting algorithm A. Algorithm A corresponds to a decision tree T where each of the $n!$ possible input permutations descends to a distinct leaf of T. (If two or more permutations descend to the same leaf, then A is not a correct sorting algorithm.) For a given permutation π, the depth of the leaf of T to which π descends corresponds to the number of comparisons made by A on input π. Thus the external path length of T is at least the sum, over all $n!$ permutations π, of the number of comparisons made by A on input π. In other words, the external path length of T is at least $n!$ times the average number of comparisons used by A. The result now follows from part (a) and the fact that $\lg(n!) = \Theta(n \lg n)$.

2(a). It is easy to check that $n_0 = 5$. Let n_1 denote the least positive integer such that $n \geq 4\sqrt{n}\ln n + 8$ holds for all $n \geq n_1$. It is easy to check that $n_1 \geq n_0$. Let c denote

$$\max\left(2, \max_{2\leq n<n_1} \frac{T(n)}{n\ln n}\right)$$

Let $P(n)$ denote the predicate "$T(n) \leq cn\ln n$". We claim that $P(n)$ holds for all $n \geq 2$. The definition of c implies that $P(n)$ holds for $2 \leq n < n_1$. Fix an integer $n \geq n_1$ and assume inductively that $P(k)$ holds for $2 \leq k < n$. To see that $P(n)$ holds, observe that

$$\begin{aligned}
T(n) &= 2T\left(\left\lfloor \frac{n}{2} + \sqrt{n}\right\rfloor\right) + n \\
&\leq 2c\left\lfloor \frac{n}{2} + \sqrt{n}\right\rfloor \ln\left\lfloor \frac{n}{2} + \sqrt{n}\right\rfloor + n \\
&\leq 2c\left(\frac{n}{2} + \sqrt{n}\right) \ln\left(\frac{n}{2} + \sqrt{n}\right) + n \\
&= 2c\left(\frac{n}{2} + \sqrt{n}\right)\left[\ln\frac{n}{2} + \ln\left(1 + \frac{2}{\sqrt{n}}\right)\right] + n \\
&\leq 2c\left(\frac{n}{2} + \sqrt{n}\right)\left(\ln n - 1 + \frac{2}{\sqrt{n}}\right) + n \\
&= cn\ln n - cn + 2c\sqrt{n} + 2c\sqrt{n}\ln n - 2c\sqrt{n} + 4c + n \\
&= cn\ln n - (c-1)n + 2c\sqrt{n}\ln n + 4c \\
&\leq cn\ln n - (c/2)n + 2c\sqrt{n}\ln n + 4c \\
&= cn\ln n - (c/2)(n - 4\sqrt{n}\ln n - 8) \\
&\leq cn\ln n,
\end{aligned}$$

where the first equality follows from the definition of $T(n)$, the first inequality follows from the induction hypothesis (note that $2 \leq \frac{n}{2} + \sqrt{n} < n$ since $n \geq n_1 \geq n_0$), the third inequality follows from $\ln(1+x) \leq x$ for $x > -1$, the fourth inequality follows from $c \geq 2$, and the last inequality holds since $n \geq n_1$ implies $n \geq 4\sqrt{n}\ln n + 8$.

2(b). $f(n) = \frac{n}{\ln n}$.

3(a). Let $B(x)$ and $C(x)$ denote the quotient and remainder polynomials when $A(x)$ is divided by $x - z$. Since $x - z$ has degree 1, $C(x)$ has degree zero, i.e., $C(x)$ is of the form $c_0x^0 = c_0$. We have $A(x) = B(x)\cdot(x-z) + C(x) = B(x)(x-z) + c_0$, and thus $A(z) = c_0$, as required.

3(b). We have $P_{k,k}(x) = (x - x_k)$ and hence $Q_{k,k}(x) = A(x) \bmod (x - x_k)$. Thus the result of part (a) implies $Q_{k,k}(x) = A(x_k)$.

It remains to prove that $Q_{0,n-1}(x) = A(x)$. We have $P_{0,n-1}(x) = \prod_{0\leq k<n}(x - x_k)$. Since $A(x)$ has degree at most $n-1$ and $P_{0,n-1}(x)$ has degree n, we deduce that $A(x)$ is the remainder when $A(x)$ is divided by $P_{0,n-1}(x)$. Thus $Q_{0,n-1}(x) = A(x)$.

3(c). Fix integers i, j, and k such that $0 \le i \le k \le j < n$. Let $B(x)$ denote the quotient polynomial when $A(x)$ is divided by $P_{i,j}(x)$. Thus $A(x) = B(x) \cdot P_{i,j}(x) + Q_{i,j}(x)$ where $P_{i,j}(x)$ has degree $j-i+1$ and $Q_{i,j}(x)$ has degree at most $j-i$. Let $C(x)$ and $D(x)$ denote the quotient and remainder polynomials when $Q_{i,j}(x)$ is divided by $P_{i,k}(x)$. Thus $Q_{i,j}(x) = C(x) \cdot P_{i,k}(x) + D(x)$ where $P_{i,k}(x)$ has degree $k-i+1$ and $D(x)$ has degree at most $k-i$. It follows that

$$\begin{aligned} A(x) &= B(x) \cdot P_{i,j}(x) + C(x) \cdot P_{i,k}(x) + D(x) \\ &= \left(B(x) \cdot P_{k+1,j}(x) + C(x)\right) P_{i,k}(x) + D(x). \end{aligned}$$

Let $E(x)$ denote the polynomial $B(x) \cdot P_{k+1,j}(x) + C(x)$. Since $A(x) = E(x) \cdot P_{i,k}(x) + D(x)$ where $P_{i,k}(k)$ has degree $k-i+1$ and $D(x)$ has degree at most $k-i$, we conclude that $E(x)$ and $D(x)$ are the quotient and remainder polynomials when $A(x)$ is divided by $P_{i,k}(x)$. Thus $D(x) = Q_{i,k}(x)$, as required.

A symmetric argument shows that $Q_{k,j}(x)$ is the quotient polynomial when $Q_{i,j}(x)$ is divided by $P_{k,j}(x)$.

3(d). Assume for simplicity that n is a power of 2. We find it easiest to describe our algorithm in terms of a complete binary tree T with n leaves indexed from 0 to $n-1$. (Such a tree can be implemented as an array.) For any node u of the tree, let $u.a$ and $u.b$ denote the minimum and maximum indices of the leaves in the subtree of T rooted at u. We perform two passes over the tree. The first pass is upward — from the leaves to the root — and the second pass is downward — from the root to the leaves.

The goal of the first pass is to compute the coefficients of the polynomial $P_{u.a,u.b}(x)$ at each node u in T. For a leaf node u, this can be done in $O(1)$ time. Now consider an internal node u with children u_L and u_R. When we come to process node u, nodes u_L and u_R have already been processed, so the coefficients of the two polynomials $P_{u_L.a,u_L.b}(x)$ and $P_{u_R.a,u_R.b}(x)$ have already been determined. Since $P_{u.a,u.b}(x)$ is the product of these two polynomials, we can use the FFT algorithm to compute the coefficients of $P_{u.a,u.b}(x)$ in $O(s \log s)$ time where $s = u.b - u.a + 1$. Let $A(n)$ denote the total running time of the first pass. Thus $A(1) = O(1)$ and $A(n) = 2A(n/2) + O(n \log n)$ for $n > 1$. Thus $A(n) = O(n \log^2 n)$

The goal of the second pass is to compute the coefficients of the polynomial $Q_{u.a,u.b}(x)$ at each node u in T. In the case where u is the root node, this is straightforward since $Q_{u.a,u.b}(x) = Q_{0,n-1}(x) = A(x)$ by the second claim of part (b). Now assume that node u is a proper descendant of the root, and let u_P denote the parent of u. When we come to process node u, node u_P has already been processed, so the coefficients of the polynomial $Q_{u_P.a,u_P.b}(x)$ have been determined. Let us assume that u is the left child of u_P; a symmetric argument holds for the case where u is the right child of u_P. As a result of the first pass, the coefficients of the polynomial $P_{u.a,u.b}(x)$ are available at nodes u_P and u. Using the result of part (c), we have $Q_{u.a,u.b}(x) = Q_{u_P.a,u_P.b}(x) \bmod P_{u.a,u.b}(x)$. Thus, using the assumption stated in the problem concerning the complexity of polynomial division, we can compute the coefficients of the polynomial $Q_{u.a,u.b}(x)$ in $O(s \log s)$ time where $s = u_P.b - u_P.a + 1$. Let $B(n)$ denote the total running time of the second pass. Thus $B(1) = O(1)$ and $B(n) = 2B(n/2) + O(n \log n)$ for $n > 1$. Thus $B(n) = O(n \log^2 n)$.

The first claim of part (b) implies that the for each k in $\{0, \ldots, n-1\}$, the value $A(x_k)$ is located at the leaf node with index k at the end of the second pass. Since $A(n)$ and $B(n)$ are each $O(n \log^2 n)$, the overall running time is $O(n \log^2 n)$, as required.

4(a). It is convenient to reindex the tasks from 1 to n in nondecreasing order of deadline, breaking ties arbitrarily.

For any integer j such that $0 \le j \le n$, let U_j denote the set of tasks $\{1, \ldots, j\}$. For any set of tasks U, we define greedy(U) as the schedule that executes the tasks in U in order of increasing index. For any set of tasks U, we define the *duration* of U as $\sum_{j \in U} e_j$, and we define the *value* of U as the value of greedy(U). We say that a schedule S for a set of tasks U is feasible if every task in U meets its deadline in S. We say that a set of tasks U is feasible if there exists a feasible schedule for U.

We claim that a set of tasks U is feasible if and only if greedy(U) is feasible. The "if" direction is immediate. To prove the "only if" direction, assume that U is feasible, and let S be a feasible schedule for U. If S is equal to greedy(U), there is nothing further to prove. Otherwise, there exist distinct tasks i and j in U such that $i > j$ and task i is scheduled immediately before task j in S. It is easy to check that if we modify schedule S by interchanging the order in which tasks i and j are scheduled to obtain a new schedule S', then S' is also a feasible schedule for U. Furthermore, the number of inversions in the permutation associated with S' is exactly one less than the number of inversions in the permutation associated with S. By repeating this argument, we eventually arrive at a feasible schedule for U with zero inversions. This feasible schedule is greedy(U), completing the proof of the claim.

Let T denote $\sum_{1 \le j \le n} e_j$. For any integers i and j such that $0 \le i \le T$ and $0 \le j \le n$, we define $a_{i,j}$ as the maximum, over all feasible subsets U of U_j with duration exactly i, of the value of greedy(U). (If no such subset U exists, we define $a_{i,j}$ as $-\infty$.) We now develop a recurrence for computing the $a_{i,j}$'s. In the following case analysis, let U be a feasible subset of U_j with duration exactly i and such that the value of greedy(U) is $a_{i,j}$.

Case 1: $i = 0$. Then U is the empty set, and hence $a_{i,j} = 0$.

Case 2: $i > 0$ and $j = 0$. Then no such U exists, and hence $a_{i,j} = -\infty$.

Case 3: $i > 0$ and $j > 0$.

Case 3.1: $d_j < i$ or $i < e_j$. Then task j cannot belong to U, and hence $a_{i,j} = a_{i,j-1}$.

Case 3.2: $e_j \le i \le d_j$. If task j belongs to U, then $a_{i,j} = v_j + a_{i-e_j,j-1}$. If task j does not belong to U, then $a_{i,j} = a_{i,j-1}$. Thus $a_{i,j} = \max(a_{i,j-1}, v_j + a_{i-e_j,j-1})$.

The total number of operations required to compute the $a_{i,j}$'s using the above recurrence is $O(nT)$. The part (a) assumption implies that $T = O(n^{c+1})$. Thus the total number of operations required to compute the desired value $a_{i,j}$'s is $O(n^{c+2})$. The desired answer is given by $\max_{0 \le i \le T} a_{i,n}$.

4(b). Let V denote $\sum_{i \le j \le n} v_j$. For any integers i and j such that $0 \le i \le V$ and $0 \le j \le n$, we define $b_{i,j}$ as the minimum, over all feasible subsets U of U_j with value exactly

i, of the duration of U. (If no such subset U exists, we define $b_{i,j}$ as ∞.) We now develop a recurrence for computing the $b_{i,j}$'s. In the following case analysis, let U be a feasible subset of U_j with value exactly i and such that the duration of greedy(U) is $b_{i,j}$.

Case 1: $i = 0$. Then U is the empty set, and hence $b_{i,j} = 0$.

Case 2: $i > 0$ and $j = 0$. Then no such U exists, and hence $b_{i,j} = \infty$.

Case 3: $i > 0$ and $j > 0$.

Case 3.1: $i < v_j$ or $d_j < e_j + b_{i-v_j,j-1}$. Suppose task j belongs to U. Then $i \geq v_j$, and hence $d_j < e_j + b_{i-v_j,j-1}$. The definition of U implies that $b_{i-v_j,j-1}$ is equal to the duration of the feasible subset $U \setminus \{j\}$ of U_{j-1}. Thus the inequality $d_j < e_j + b_{i-v_j,j-1}$ implies that task j misses its deadline in greedy(U), contradicting the feasibility of U. We conclude that task j does not belong to U, and hence that $b_{i,j} = b_{i,j-1}$.

Case 3.2: $i \geq v_j$ and $e_j + b_{i-v_j,j-1} \leq d_j$. If task j does not belong to U, then $b_{i,j} = b_{i,j-1}$. If task j belongs to U, then $b_{i,j} = e_j + b_{i-v_j,j-1}$. Thus $b_{i,j} = \min(b_{i,j-1}, e_j + b_{i-v_j,j-1})$.

The total number of operations required to compute the $b_{i,j}$'s using the above recurrence is $O(nV)$. The part (b) assumption implies that $V = O(n^{c+1})$. Thus the total number of operations required to compute the desired value $b_{i,j}$'s is $O(n^{c+2})$. The desired answer is given by the maximum value of i for which $b_{i,n} < \infty$.